

Essential C++ 中文版

[美] **Stanley B. Lippman** 著
侯捷 译

Essential C++

Stanley B. Lippman

C++ In-Depth Series • Bjarne Stroustrup

Essential C++



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Essential C++ 中文版

“拿起这本书，你可以在短时间内熟悉C++。Stan选择了范围广泛而又复杂的一些主题，把它们的门槛调低到C++新手开发真正程序所需要的最基本层次。本书以实例引导学习，这种做法很有效，让我们得以顺利学完整个课程。”

——Steve Vinoski, IONA

对于那些已经开始从事软件设计，抽不出太多时间学习新技术的程序员，*Essential C++*提供了一条捷径，循此可以在短时间内把C++应用到你的工作上。本书强调C++编程过程中一定会遭遇的要素，以及可协助解决实际问题的技术。

本书以四个面向来表现C++的本质：*procedural*（面向过程的）、*generic*（泛型的）、*object-based*（基于对象的）、*object-oriented*（面向对象的）。本书的组织围绕着一系列逐渐繁复的程序问题，以及用以解决这些问题的语言特性。循此方式，你将不只学到C++的功能和结构，也可学习到它们的设计目的和基本原理。

你可以从本书中发现以下关键主题：

- Generic（泛型）编程风格和Standard Template Library（STL）
- Object-based（基于对象）编程风格和class的设计
- Object-oriented（面向对象）编程风格和class层次体系的设计
- Function template和class template的设计和运用
- Exception handling（异常处理）与运行时类型辨识（Run-Time Type Identification）

此外，两份附录极具价值。附录A提供了每章最后所列之练习题的完整解答和详细说明。附录B提供了一份泛型算法快速参考手册（含运用实例）。

这本言简意赅的教科书可带给你C++基本知识，为未来建立个人专业技术打下基础。

作者简介

Stanley B. Lippman是梦工厂电影动画公司的核心技术组成员。加入梦工厂之前，Stan是迪斯尼电影动画公司的主要工程师。更早之前他在贝尔实验室领导过Cfront 3.0和2.1的编译器开发团队。他是Bjarne Stroustrup所领导的贝尔实验室基础项目中的一员。Stan是*C++ Primer*及*Inside the C++ Object Model*的作者，这些极为成功的图书均由Addison-Wesley出版。他还是*C++ Gems*的编辑（此书由Cambridge University Press出版）。他的工作成果应用于多部电影中，包括*Hunchback of Notre Dame*和*Fantasia 2000*。

译者简介

侯捷是信息教育者，写译书籍，主笔专栏，培训业界人员，并于南京大学、同济大学开课。进入教育领域前，曾担任台湾工业研究院机械所和电通所的副研究员与特约研究员，分别研发CAD/CAM软件和Windows多媒体系统。捷是《深入浅出MFC》和《STL源码剖析》的作者，也是*C++ Primer*和*Inside the C++ Object Model*两书中文版译者。



@博文视点Broadview



PEARSON

www.pearson.com



策划编辑：张春雨 @永恒的侠少
责任编辑：白涛

上架建议：程序语言

ISBN 978-7-121-20934-5



9 787121 209345 >

定价：65.00元

Essential C++

中文版

[美] Stanley B. Lippman 著

侯捷 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

本书以四个面向来表现 C++ 的本质: procedural (面向过程的)、generic (泛型的)、object-based (基于对象的)、object-oriented (面向对象的)。全书围绕一系列逐渐繁复的程序问题,以及用以解决这些问题的语言特性来组织。循此方式,你将不只学到 C++ 的功能和结构,也可学到它们的设计目的和基本原理。

本书适合那些已经开始从事软件设计,又抽不出太多时间学习新技术的程序员阅读。

Authorized translation from the English language edition,entitled ESSENTIAL C++,1E,9780201485189 by LIPPMAN,STANLEY B.,published by Pearson Education,Inc.,Publishing as Addison-Wesley Professional,Copyright©2000 Pearson Education,Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright ©2013.

本书简体中文版专有版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2013-4126

图书在版编目(CIP)数据

Essential C++中文版 / (美)李普曼(Lippman, S. B.)著;侯捷译.——北京:电子工业出版社,2013.8

ISBN 978-7-121-20934-5

I. ①E... II. ①李... ②侯... III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 150792 号

责任编辑:白涛

印刷:三河市双峰印刷装订有限公司

装订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开本:787×980 1/16 印张:18.75 字数:440 千字

印次:2013 年 8 月第 1 次印刷

定价:65.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010)88258888。

悦读上品 得乎益友

孔子云：“取乎其上，得乎其中；取乎其中，得乎其下；取乎其下，则无所得矣”。

对于读书求知而言，这句古训教我们去读好书，最好是好书中的上品——经典书。其中，科技人员要读的技术书，因为直接关乎客观是非与生产效率，阅读选材本更应慎重。然而，随着技术图书品种的日益丰富，发现经典书越来越难，尤其对于涉世尚浅的新读者，更为不易，而他们又往往是最需要阅读、提升的重要群体。

所谓经典书，或说上品，是指选材精良、内容精练、讲述生动、外延丰盈、表现手法体贴入微的读品，它们会成为读者的知识和经验库中的重要组成部分，并且拥有从不断重读中汲取养分的空间。因此，选择阅读上品的问题便成了有效阅读的首要问题。当然，这不只是效率问题，上品促成的既是对某一种技术、思想的真正理解和掌握，同时又是一种感悟或享受，是一种愉悦。

与技术本身类似，经典 IT 技术书多来自国外。深厚的积累、良好的写作氛围，使一批大师为全球技术学习者留下了璀璨的智慧瑰宝。就在那个年代即将远去之时，无须回眸，也能感受到这一部部厚重而深邃的经典著作，在造福无数读者后从未蒙尘的熠熠光辉。而这些凝结众多当今国内技术中坚美妙记忆与绝佳体验的技术图书，虽然尚在国外图书市场上大放异彩，却已逐渐淡出国人的视线。最为遗憾的是，迟迟未有可以填补空缺的新书问世。而无可替代，不正是经典书被奉为主臬的原因？

为了不让国内读者，尤其是即将步入技术生涯的新一代读者，就此错失这些滋养过先行者们的好书，以出版 IT 精品图书，满足技术人群需求为己任的我们，愿意承担这一使命。本次机遇惠顾了我们，让我们有机会携手权威的 Pearson 公司，精心推出“传世经典书丛”。

在我们眼中，“传世经典”的价值首先在于——既适合喜爱科技图书的读者，也符合专家们挑剔的标准。幸运的是，我们的确找到了这些堪称上品的佳作。丛书带给我们的幸运颇多，细数一下吧。

得以引荐大师著作

有恐思虑不周，我们大量参考了国外权威机构和网站的评选结果，并得到了 Pearson 的专业支持，又进

一步对符合标准之图书的国内外口碑与销售情况进行细致分析,也听取了国内技术专家的宝贵建议,才有幸选出对国内读者最富有技术养分的大师上品。

向深邃的技术内涵致敬

中外技术环境存在差异,很多享誉国外的好书未必适用于国内读者;且技术与应用瞬息万变,很容易让人心生迷惘或疲于奔命。本丛书的图书遴选,注重打好思考方法与技术理念的根基,旨在帮助读者修炼内功,提升境界,将技术真正融入个人知识体系,从而可以一通百通,从容面对随时涌现的技术变化。

翻译与评注的双项选择

引进优秀外版著作,将其翻译为中文供国内读者阅读,较为有效与常见。但另有一些外语水平较高、喜好评读原版的读者,苦于对技术理解不足,不能充分体会原文表述的精妙,需要有人指导与点拨。而一批本土技术精英经过长期经典熏陶及实践锤炼,已足以胜任这一工作。有鉴于此,本丛书在翻译版的同时推出融合英文原著与中文点评、注释的评注版,供不同志趣的读者自由选择。

承蒙国内一流译(注)者的扶持

优秀的英文原著最终转化为真正的上品,尚需跨越翻译鸿沟,外版图书的翻译质量一直屡遭国内读者诟病。评注版的增值与含金量,同样依赖于评注者的高卓才具。好在,本丛书得到了久经考验的权威译(注)者的认可和支持,首肯我们选用其佳作,或亲自参与评注工作。正是他们的参与保证了经典的品质,既再次为我们的选材把关,更提供了一流的中文表述。

期望带给读者良好的阅读体验

一本好书带给人的愉悦不止于知识收获,良好的阅读感受同样不可缺少,且对学业不无助益。为了让读者收获与上品相称的体验,我们在图书装帧设计与选材用料上同样不敢轻率,惟愿送到读者手中的除了珠玑章句,还有舒适与熨帖的视觉感受。

所有参与丛书出版的人员,尽管能力有限,却无不心怀严谨之心与完美愿望。如果读者朋友能从潜心阅读这些上品中偶有获益,不啻为对我们工作的最佳褒奖。若有阅读感悟,敬请拨冗告知,以鼓励我们继续在这一道路上贡献绵薄之力。如有不周之处,也请不吝指教。

满汉全席之外 (译序/侯捷)

Stanley B. Lippman 所著的 C++ *Primer* 雄踞书坛历久不衰, 堪称 C++ 最佳教科书。但是走过十年头之后, 继 1237 页的 C++ *Primer* 第 3 版, Lippman 又返璞归真地写了这本 276 页的 *Essential C++*。有了满汉全席, 为何还眷顾清粥小菜? 完成了伟大的巨著, 何必回头再写这么一本轻薄短小的初学者用书呢?

所有知道 Lippman 和 C++ *Primer* 的人, 脸上都浮现相同的问号。

轻薄短小并不是判断适合初学与否的依据。Lippman 写过 *Inside the C++ Object Model*, 280 页小开本, 崩掉多少 C++ 老手的牙。本书之所以号称适合初学者, 不在于轻薄短小, 在于素材选择与组织安排。

关于 Lippman 重作冯妇的故事, 他自己在前言中有详细的介绍。他的转折, 他的选择, 他的职责, 乃至于这本书的纲要和组织, 前言中都有详细的交待。这方面我不必再置一词。

身为 C++ *Primer*, 3rd Edition 的译者, 以及多本进阶书籍的作者, 我必须努力说服自己, 才能心甘情愿地将精力与时间用来重复过去的足迹。然而, 如果连 Lippman 都愿意为初学者再铺一条红地毯, 我也愿意为初学者停留一下我的脚步。

我是一名信息教育者, 写译书籍, 培训人员, 在大学开课……。我真正第一线面对大量学习者。借此机会我要表达的是, 所谓“初学者”实在是个过于笼统的名词与分类(呃, 谈得上分类吗)。一般所谓“初学者”, 多半想象是大一新生程度。其实 C++ 语言存在各种“初学者”, 有 13 岁的, 有 31 岁的(当然也有 41 岁的)。只要是第一次接触这个语言, 就是这个语言的初学者, 他可能才初次接触计算机, 可能浸淫 Pascal/C 语言十年之久, 也可能已有 Smalltalk/Java 三年经验。有人连计算机基本概念都没有, 有人已经是经验丰富的软件工程师。这些人面对 C++, 学习速度、教材需求、各人领悟, 相同吗?

大不同矣！

每个人都以自己的方式来诠释“初学者”这个字眼，并不经意地反映出自己的足迹。初学者有很多很多种，“初学者”一词却无法反映他们的真实状态。

固然，轻薄短小的书籍乍见之下让所有读者心情轻松，但如果舍弃太多应该深入的地方不谈，也难免令人行止失据，进退两难。这本小书可以是你的起点，但绝不能够是你的终点。

作为一本优秀教科书，轻薄短小不是重点，素材选择与组织安排，表达的精准与阅读的顺畅，才是重点。

作为一个好的学习者，背景不是重点，重要的是，你是否具备正确的学习态度。起步固然可从轻松小品开始，但如果碰上大部头巨著就退避三舍逃之夭夭，面对任何技术只求快餐速成，学编程语言却从来不编写程序，那就绝对没有成为高手乃至专家的一天。

有些人的学习，自练一身铜筋铁骨，可以在热带丛林中披荆斩棘，在莽莽草原中追奔逐北。有些人的学习，既未习惯大部头书，也未习惯严谨格调，更未习惯自修勤学，是温室里的一朵花，没有自立自强的本钱。

章节的安排，篇幅的份量，索引的保留，习题加解答，以及网上的服务，都使这本小书成为自修妙品、C++专业课程的适当教材。诚挚希望《Essential C++中文版》的完成，帮助更多人获得 C++ 的学习乐趣——噢，是的，OOP（面向对象编程）可以带给你很多乐趣，我不骗你 ☺

侯捷

2012/09/22

敬请注意：

1. 本书与英文版页页对译，从而得以保留原书索引。
2. 本书附加“中英术语对照表”于附录 C，并于其中说明中英术语的采用原则。

前言

Preface

天啊，这本书竟是如此轻薄短小。我真想大叫一声“哇欧”！C++ *Primer* 加上索引、扉页、谢词之后，厚达 1237 页，而此书却只有薄薄 276 页。套用拳击术语，这是一部“轻量级”作品。

每个人都会好奇这究竟是怎么回事。的确，这里头有一段故事。

过去数年来，我不断缠着迪斯尼电影动画公司（Disney Feature Animation）的每一个人，请求让我亲身参与一部电影的制作。我缠着导演，甚至 Mickey 本人（如果我可以说不出的话），要求一份管理工作。我会如此疯狂，部分原因是深陷于好莱坞大屏幕那令人神往的无尽魔力而难以自拔。除了计算机科学方面的学位，我还拥有艺术硕士的头衔，而电影工作似乎可以为我带来个人专长的某种整合。我要求管理工作，为的是从制片过程中获取经验，以便提供实际有用的工具。身为一个 C++ 编译器编写者，我一直都是自己最主要的用户之一。而你知道，当你是自己软件的主力抱怨者时，你就很难再为自己辩护或觉得受到不公平的责难。

《幻想曲 2000》（*Fantasia 2000*）片中有一段火鸟（Firebird）的特效镜头。其计算机特效指导对于我的加盟颇感兴趣。不过，为了掂掂我的斤两，他要求我先写个工具，读入为某段场景所摄的原始数据，再由此产生可嵌入 Houdini 动画套件中的摄影机节点（camera node）。当然，我用 C++ 把它顺利搞定了。他们爱死它了，我也因此得到了我梦寐以求的工作。

有一次，在制片过程中（在此特别感谢 Jinko 和 Chyuan），我被要求以 Perl 重写那个工具。其他的 TD 并非编程高手，仅仅知道 Perl、Tcl 之类的程序语言。（TD 是电影界的术语，指的是技术导演。我是这部片子的软件 TD，我们还有一位灯光 TD [你好，Mira]，一位模型 TD [你好，Tim]，以及电影特效动画师 [你好，Mike, Steve, Tonya]。）而且，喔，天啊，我得赶着点，因为我们想要获得一些观念上的实证，而导演（你好，Paul 和 Gaetan）及特效总监（你好，Dave）正等着结果，准备呈给公司大头目（你好，Peter）。这虽然不是什么紧急要务，可是，你知道的……，唉。

这令我感到些许为难。我自信可以用 C++ 快速完成，但我不懂 Perl。好吧，我想，我去找本书抱抱佛脚好了——前提是这本书不能太厚，起码此刻不能太厚。而且它最好不要告诉我太多东西，虽然我知道我应该知道每一样东西，不过暂且等等吧。毕竟这只是一场表演：导演们需要一些经过证实的概念，艺术家需要一些东西协助证实其概念，而制片（你好，heck），她需要的是一天 48 小时。此刻我不需要全世界最棒的 Perl 大全，我需要的是一本能妥善引导我前进，使我不致偏离正轨过远的小书。

我找到了 Randal Schwartz 的 *Learning Perl*，它让我立即上手并进展神速，而且颇具阅读趣味。不过，就像其他有趣的计算机书籍一样，它也略去了不少值得一读的内容——尽管在那个时间点，我并不需要了解所有内容，我只需要让我的 Perl 程序乖乖动起来。

我终于在感伤的心境中明白，C++ *Primer* 第三版其实无法扮演人们在初学 C++ 时的导师角色。它太庞大了。当然，我还是认为它是一本让我骄傲的巨著——特别是由于邀请到 Josée Lajoie 共同完成。但是，对于想立刻学会 C++ 程序语言的人来说，这本巨著实在过于庞大复杂。这正是本书的由来。

你或许会想，C++ 又不是 Perl。完全正确！本书也非 *Learning Perl*，它谈的是如何学习 C++。真正的问题在于，谁能够在散尽千页篇幅之后，犹敢自称教导了所有的东西呢？

1. **精细度。**在计算机绘图领域中，精细度指的是影像被描绘出来的鲜明程度。画面左上角那位骑在马背上的匈奴人，需要一张看得清楚眼睛的脸、头发、五点钟方向的影子、衣服……。匈奴人的背后——不，不是那块岩石，老天——唔，相较之下无关紧要。因此我们不会以相同的精细度来描绘这两个影像。同样道理，本书的精细度在相当程度上做了降低。依我看，C++ *Primer* 除了在运算符重载（operator overloading）方面的实例讨论稍嫌不足外，可说极其完备了（我敢这么说是因为 Josée 也有一份功劳）。但尽管如此，C++ *Primer* 还花了 46 页篇幅讨论操作符重载，并附上了范例，而本书却仅以两页带过。
2. **语言核心。**当我还是 C++ *Report* 的编辑时，我常说，杂志编辑有一半工作花在决定哪些题材应该放入，哪些不要。这句话对本书一样成立。本书内容是围绕在编程过程中所发生的一系列问题组织的。我介绍编程语言本身的特性，借此来为不同的问题提供解决之道。书中并未提及任何一个可由多继承或虚继承解决的问题，所以我也就完全没有讨论这两个主题。然而，为了实现 iterator class，我必须引入嵌套类型（nested type）。Class 的类型转换操作符很容易被错用，解释起来也很复杂，所以我不打算在书中提到它。诸如此类。我对题材的选择以及对语言特性的呈现顺序，欢迎大家指教批评。这是我的选择，也是我的职责。
3. **范例的数量。**C++ *Primer* 有数百页代码，巨细靡遗，其中甚至包括一套面向对象的（Object Oriented）文本检索系统，以及十个左右的完整 class。虽然本书也有代码，但数量远不及 C++ *Primer*。为了弥补这项缺憾，我将所有习题解答都置于附录 A。诚如我的编辑 Deborah Lafferty 所言，“如果你想提高教学速度，唾手可得的解答对于学习的强化极有帮助。”

结构与组织

本书由七章和两份附录构成。第 1 章借着撰写一个具有互动性质的小程序，描绘 C++ 语言预先定义的部分。这一章涵盖了内置的数据类型、语言预定义的运算符（operator）、标准库中的 `vector` 和 `string`、条件语句和循环语句、输入和输出用的 `iostream` 库。我之所以在本章介绍 `vector` 和 `string` 这两个 class，是因为我想鼓励读者多多利用它们取代语言内置的数组（array）和 C-style 字符串。

第 2 章解释函数的设计与使用，并逐一查看 C++ 函数的多种不同风貌，包括 inline 函数、重载（overloaded）函数、function template，以及函数指针（pointers to functions）。

第 3 章涵盖了所谓的 Standard Template Library（STL）：一组容器类（包括 `vector`、`list`、`set`、`map`，等等）、一组作用于容器上的泛型算法（包括 `sort()`、`copy()`、`merge()`，等等）。附录 B 按字典顺序列出了最广为运用的泛型算法，并逐一附上了使用实例。

身为一个 C++ 程序员，你的主要任务便是提交 class 以及面向对象的 class 层次体系。第 4 章将带领你亲身了解 class 机制的设计与使用过程。在这个过程中，你会看到如何为自身的应用系统建立起专属的数据类型。第 5 章介绍如何扩展 class，使多个相关的 class 形成族系，支持面向对象的 class 层次体系。以我在梦工厂动画电影公司（Dreamworks Animation）担任顾问的经验为例，那时我们设计了一些 class，用来进行四个频道影像合成之类的工作。我们使用了继承和动态绑定（dynamic binding）技术，定义影像合成所需的 class 层次体系，而不只是设计八个独立的 class。

第 6 章的重头戏是 class template，那是建立 class 时的一种先行描述，让我们得以将 class 用到的（或多个）数据类型或数据值，抽离并参数化。以 `vector` 为例，可能需要将其元素的类型加以参数化，而 `buffer` 的设计不仅得将元素类型参数化，还得将其缓冲区容量参数化。本章的行进路线围绕在二分树（binary tree）class template 的实现上。

最后一章，第 7 章，介绍如何使用 C++ 的异常处理机制（exception handling facility），并示范如何将它融入标准库所定义的异常体系中。附录 A 是本书习题解答。附录 B 提供了关于最广为运用的一些泛型算法的相关讨论与使用实例。

关于源代码

本书的所有程序，以及习题解答中的完整代码，都可从网上获得。你可以在 Addison Wesley Longman 的网站（www.awl.com/cseng/titles/0-201-48518-4）或我的个人首页（www.objectwrite.com）中取得。所有程序均在 Visual C++ 5.0 环境中以 Intel C++ 编译器测试过，也在 Visual C++ 6.0 环

境中以 Microsoft C++ 编译器测试过。你或许需要稍微修改一下代码才能在自己的系统上编译成功。如果你需要做一些修改并且做了，请将修改结果寄一份给我 (slippman@objectwrite.com)，我会将它们附上你的大名，附于习题解答代码中。请注意，本书并未显现所有代码。

致谢

在这里，我要特别感谢 C++ *Primer* 第三版的共同作者 Josée Lajoie。不仅因为她为本书初稿提供了许多深入见解，更因为她在背后不断地给我鼓舞。我也要特别感谢 Dave Slayton 以他那犀利的绿色铅笔，彻底审阅了文本内容与程序范例。Steve Vinoski 则以同情但坚决的口吻，为本书初稿提供了许多宝贵意见。

特别感谢 Addison-Wesley 编辑团队：全书编辑 Deborah Lafferty 从头到尾支持这个项目；审稿编辑 Besty Hardinger 对本书文字的可读性贡献最大；产品经理 John Fuller 带领我们把一堆文稿化为一本完整的图书。

撰写本书的过程中，我同时还担任独立顾问工作，因此必须兼顾书稿和客户。感谢我的客户对我如此体谅和宽容。我要感谢 Colin Lipworth、Edwin Leonard、Kenneth Meyer，因为你们的耐心与信赖，本书才得以完成。

更多读物

内举不避亲，我要推荐 C++ 书籍中最好的两本，那便是 Lippman 与 Lajoie 合著的 C++ *Primer*，以及 Stroustrup 的著作 *The C++ Programming Language*。这两本书目前均为第 3 版。我会在本书各主题内提供其他更深入的参考书目。以下是本书的参考书目。（你可以在 C++ *Primer* 和 *The C++ Programming Language* 中找到更广泛的参考文献。）

[LIPPMAN98] Lippman, Stanley and Josée Lajoie, *C++ Primer, 3rd Edition*, Addison Wesley Longman, Inc., Reading, MA (1998) ISBN 0-201-82470-1.

[LIPPMAN96a] Lippman, Stanley, *Inside the C++ Object Model*, Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-83454-5.

[LIPPMAN96b] Lippman, Stanley, Editor, *C++ Gems*, a SIGS Books imprint, Cambridge University Press, Cambridge, England (1996) ISBN 0-13570581-9.

[STROUSTRUP97] Stroustrup, Bjarne, *The C++ Programming Language, 3rd Edition*, Addison Wesley Longman, Inc., Reading, MA (1997) ISBN 0-201-88954-4.

[SUTTER99] Sutter, Herb, *Exceptional C++*, Addison Wesley Longman, Inc., Reading, MA (2000) ISBN 0-201-61562-2.

排版约定

本书字体（英文版）为 10.5 pt Palatino。代码和语言关键字为 8.5 pt `lucida`。书中出现的标识符如果后面紧接着 C++ 的 function call 运算符（也就是一对圆括号 `()`），即代表某个函数名称。因此，`foo` 代表程序中的某个 object，`bar()` 代表程序中的某个函数。各个 class 的名称以 Palatino 字形呈现。

简体中文版的排版约定是：内文中的一般英文字为 9 pt Times New Roman。代码和语言关键字为 8 pt Courier New。各个 class 的名称亦为 8 pt Courier New。异常类（exception class）以 8 pt Lucida Sans 呈现。英文长术语（例如 template parameter list, by reference, exception safe）采用 8 pt Arial。运算符名称采用 9 pt Footlight MT Light。译注均以楷体呈现。

目录

Contents

满汉全席之外（译序/侯捷）	v
前言 Preface	xi
结构与组织	xiii
关于源代码	xiii
致谢	xiv
更多读物	xiv
排版约定	xv
第 1 章 C++编程基础 Basic C++ Programming	1
1.1 如何撰写 C++程序	1
1.2 对象的定义与初始化	7
1.3 撰写表达式	10
1.4 条件语句和循环语句	15
1.5 如何运用 Array 和 Vector	22
1.6 指针带来弹性	26
1.7 文件的读写	30
第 2 章 面向过程的编程风格 Procedural Programming	35
2.1 如何编写函数	35
2.2 调用函数	41
2.3 提供默认参数值	50
2.4 使用局部静态对象	53
2.5 声明 inline 函数	55

2.6 提供重载函数.....	56
2.7 定义并使用模板函数.....	58
2.8 函数指针带来更大的弹性.....	60
2.9 设定头文件.....	63
第 3 章 泛型编程风格 Generic Programming	67
3.1 指针的算术运算.....	68
3.2 了解 Iterator (泛型指针)	73
3.3 所有容器的共通操作.....	76
3.4 使用顺序性容器.....	77
3.5 使用泛型算法.....	81
3.6 如何设计一个泛型算法.....	83
3.7 使用 Map	90
3.8 使用 Set.....	91
3.9 如何使用 Iterator Inserter.....	93
3.10 使用 iostream Iterator	95
第 4 章 基于对象的编程风格 Object-Based Programming	99
4.1 如何实现一个 Class	100
4.2 什么是构造函数和析构函数.....	104
4.3 何谓 mutable (可变) 和 const (不变)	109
4.4 什么是 this 指针.....	113
4.5 静态类成员.....	115
4.6 打造一个 Iterator Class	118
4.7 合作关系必须建立在友谊的基础上.....	123
4.8 实现一个 copy assignment operator.....	125
4.9 实现一个 function object.....	126
4.10 重载 iostream 运算符.....	128
4.11 指针, 指向 Class Member Function.....	130
第 5 章 面向对象编程风格 Object-Oriented Programming	135
5.1 面向对象编程概念.....	135
5.2 漫游: 面向对象编程思维.....	138
5.3 不带继承的多态.....	142

5.4	定义一个抽象基类.....	145
5.5	定义一个派生类.....	148
5.6	运用继承体系.....	155
5.7	基类应该多么抽象.....	157
5.8	初始化、析构、复制.....	158
5.9	在派生类中定义一个虚函数.....	160
5.10	运行时的类型鉴定机制.....	164
第 6 章 以 template 进行编程 Programming with Templates		167
6.1	被参数化的类型.....	169
6.2	Class Template 的定义	171
6.3	Template 类型参数的处理.....	172
6.4	实现一个 Class Template	174
6.5	一个以 Function Template 完成的 Output 运算符.....	180
6.6	常量表达式与默认参数值.....	181
6.7	以 Template 参数作为一种设计策略	185
6.8	Member Template Function	187
第 7 章 异常处理 Exception Handling		191
7.1	抛出异常.....	191
7.2	捕获异常.....	193
7.3	提炼异常.....	194
7.4	局部资源管理.....	198
7.5	标准异常.....	200
附录 A 习题解答 Exercises Solutions.....		205
附录 B 泛型算法参考手册 Generic Algorithms Handbook		255
附录 C 中英术语对照 侯捷		271
	英文术语的采用原则.....	271
	中英术语对照（按字母顺序排列）	272
索引 Index.....		277

C++ 编程基础

Basic C++ Programming

本章，我们将从一个小程序开始，通过它来练习 C++ 程序语言的基本组成。其中包括：

1. 一些基础数据类型：布尔值 (Boolean)、字符 (character)、整数 (integer)、浮点数 (floating point)。
2. 算术运算符、关系运算符以及逻辑运算符，用以操作上述基础数据类型。这些运算符不仅包括一般常见的加法运算符、相等运算符 (==)、小于等于 (<=) 运算符以及赋值 (assignment, =) 运算符，也包含比较特殊的递增 (++) 运算符、条件运算符 (?:)，以及复合赋值 (+= 等) 运算符。
3. 条件分支和循环控制语句，例如 if 语句和 while 循环，可用来改变程序的控制流程。
4. 一些复合类型，例如指针及数组。指针可以让我们间接参考一个已存在的对象，数组则用来定义一组具有相同数据类型的元素。
5. 一套标准的、通用的抽象化库，例如字符串和向量 (vector)。

1.1 如何撰写 C++ 程序

How to Write a C++ Program

此刻，假设我们需要撰写一个简易程序，必须能够将一段信息送至用户的终端 (terminal)。信息的内容则是要求用户输入自己的名字。然后程序必须读取用户所输入的名字，将这个名字储存起来，以便后续操作使用。最后，送出一个信息，以指名道姓的方式向用户打招呼。

那么，该从何处着手呢？每个 C++ 程序都是从一个名为 main 的函数开始执行，我们就从这个地方着手吧！main 是个由用户自行撰写的函数，其通用形式如下：

```
int main()
{
    // 我们的程序代码置于此处
}
```

`int` 是 C++ 程序语言的关键字。所谓关键字 (keyword)，就是程序语言预先定义的一些具有特殊意义的名称。`int` 用来表示语言内置的整数数据类型。(下一节我将针对数据类型做更详细的说明。)

函数 (function) 是一块独立的程序代码序列 (code sequence)，能够执行一些运算。它包含四个部分：返回值类型 (return type)、函数名称、参数列表 (parameter list)，以及函数体 (function body)。下面依次简要介绍每一部分。

函数的返回值通常用来表示运算结果。`main()` 函数返回整数类型。`main()` 的返回值用来告诉调用者，这个程序是否正确执行。习惯上，程序执行无误时我们令 `main()` 返回零。若返回一个非零值，表示程序在执行过程中发生了错误。

函数的名称由程序员选定。函数名最好能够提供某些信息，让我们容易了解函数实际上在做些什么。举例来说，`min()` 和 `sort()` 便是极佳的命名。`f()` 和 `g()` 就没有那么好了。为什么？因为后两个名称相形之下无法告诉我们函数的实际执行操作。

`main` 并非程序语言定义的关键字。但是，执行我们这个 C++ 程序的编译系统，会假设程序中定义有 `main()` 函数。如果我们没有定义，程序将无法执行。

函数的参数列表 (parameter list) 由两个括号括住，置于函数名之后。空的参数列表，如 `main()`，表示函数不接受任何参数。

参数列表用来表示“函数执行时，调用者可以传给函数的类型列表”。列表之中以逗号隔开各个类型（通常我们会说用户“调用 (call 或是 *invoke*)”某个函数)。举例来说，如果我们编写 `min()` 函数，使其返回两数中较小者，那么它的参数列表应该注明两个即将被拿来比较的数值的类型。这样一个用来比较两整数值的 `min()` 函数，可能会以如下形式定义：

```
int min(int val1, int val2)
{
    // 程序代码置于此处
}
```

函数的主体 (body) 由大括号 ({ }) 标出，其中含有“提供此函数之运算”的程序代码。双斜线 (//) 表示该行内容为注释，也就是程序员对程序代码所做的某些说明。注释的撰写是为了便于阅读者更容易理解程序。编译过程中，注释会被忽略掉。双斜线之后直至行末的所有内容，都会被当作程序注释。

我们的第一件工作就是要将信息送至用户终端。数据的输入与输出，并非 C++ 程序语言本身定义的一部分（此精神同 C 语言，见 K&R 第 7 章），而是由 C++ 的一套面向对象的类层次体系 (classes hierarchy) 提供支持，并作为 C++ 标准库 (standard library) 的一员。

所谓类 (class)，是用户自定义的数据类型 (user-defined data type)。class 机制让我们得以将数据类型加入我们的程序中，并有能力识别它们。面向对象的类层次体系 (class hierarchy) 定义了整个家

族体系的各相关类型，例如终端与文件输入设备、终端与文件输出设备等。（关于类及面向对象的程序设计（object-oriented programming）这两个课题，本书还有许多篇幅会涉及。）

C++ 事先定义了一些基础数据类型：布尔值（Boolean）、字符（character）、整数（integer）、浮点数（floating point）。虽然它们为我们的编程任务提供了基石，但它们并非程序的重心所在。举个例子，照相机具有一个性质：空间位置。这个位置通常可以用三个浮点数表示。照相机还具备另一个性质：视角方向，同样也可以用三个浮点数表示。通常我们还会用所谓 aspect ratio 来描述照相机窗口的宽/高比，这只需单一浮点数即可表示。

最原始最基本的情况下，照相机可以用七个浮点数来表示，其中六个分别组成了两组 x、y、z 坐标。以这么低级的方式来进行编程，势必会让我们的思考不断地在“照相机抽象性质”和“相应于照相机的七个浮点数”之间反复来回。

class 机制，赋予了我们“增加程序内之类型抽象化层次”的能力。我们可以定义一个 Point3d class，用来表示“空间位置”和“视角方向”两个性质。同样的道理，我们可以定义一个 Camera class，其中包含两个 Point3d 对象和一个浮点数。以这种方式，我们同样使用七个浮点数来表示照相机的性质，不同的是我们的思考不再直接面对七个浮点数，而是转为对 Camera class 的操作。

class 的定义，一般来说分为两部分，分别写在不同的文件中。其中之一是所谓的“头文件（header file）”，用来声明该 class 所提供的各种操作行为（operation）。另一个文件，程序代码文件（program text），则包含了这些操作行为的实现内容（implementation）。

欲使用 class，我们必须先在程序中包含其头文件。头文件可以让程序知道 class 的定义。C++ 标准的“输入/输出库”名为 iostream，其中包含了相关的整套 class，用以支持对终端和文件的输入与输出。我们必须包含 iostream 库的相关头文件，才能够使用它：

```
#include <iostream>
```

我将利用已定义好的 cout（读作 see out）对象，将信息写到用户的终端中。output 运算符（<<）可以将数据定向到 cout，像下面这样：

```
cout << "Please enter your first name: ";
```

上述这行便是 C++ 所谓的“语句（statement）”。语句是 C++ 程序的最小独立单元。就像自然语言中的句子一样。语句以分号作为结束。以上语句将常量字符串（string literal，封装于双引号内）写到了用户的终端。在这之后，用户便会看到如下信息：

```
Please enter your first name:
```

接下来我们要读取用户的输入内容。读取之前，我们必须先定义一个对象，用以储存数据。欲定义一个对象，必须指定其数据类型，再给定其标识符。截至目前，我们已经用过 `int` 数据类型。但是要用它来储存某人的名字，几乎是不可能的事。更适当的数据类型是标准库中的 `string class`：

```
string user_name;
```

如此一来我们便定义了一个名为 `user_name` 的对象，它属于 `string class`。这样的定义有个特别的名称，称为“声明语句 (declaration statement)”。单只写下这行语句还不行，因为我们还必须让程序知道 `string class` 的定义。因此还必须在程序中包含 `string class` 的头文件：

```
#include <string>
```

接下来便可利用已定义好的 `cin` (读作 `see in`) 对象来读取用户在终端上的输入内容。通过 `input` 运算符 (`>>`) 将输入内容定向到具有适当类型的对象身上：

```
cin >> user_name;
```

以上所描述的输出和输入操作，在用户终端上显示如下 (输入部分以粗体表示)：

```
Please enter your first name: anna
```

剩下的工作就是打印出向用户打招呼的信息了。我们希望获得下面这样的输出结果：

```
Hello, anna ... and goodbye!
```

当然，这样打招呼稍嫌怠慢。但这不过才第1章而已。本书结束之前我会有更具创意的招呼方式。

为了产生上述信息，我们的第一个步骤便是将输出位置 (屏幕上的光标) 调到下一行起始处。将换行 (`newline`) 字符常量写至 `cout`，便可达到这个目的：

```
cout << '\n';
```

所谓字符常量 (`character literal`) 系由一组单引号括住。字符常量分为两类：第一类是可打印字符，例如英文字母 (`'a'`、`'A'`，等等)、数字、标点符号 (`'.'`、`'-'`，等等)。另一类是不可打印字符，例如换行符 (`'\n'`) 或制表符 (`tab`, `'\t'`)。由于不可打印字符并无直接的表示法 (这表示我们无法使用单一而可显示的字符来独立表示)，所以必须以两个字符所组成的字符序列来表示。

现在，我们已经将输出位置调整到下一行起始处，接着要产生 `Hello` 信息：

```
cout << "Hello, ";
```

接下来应该在此处输出用户的名字。这个名字已经储存在 `user_name` 这个 `string` 对象中。我们应当如何进行呢？其实就和处理其他数据类型一样，像下面这样即可：

```
cout << user_name;
```


最后我们以道别来结束这段招呼信息（注意，字符串常量内可以同时包含可打印字符和不可打印字符）：

```
cout<< " ... and goodbye!\n";
```

一般而言，所有内置数据类型都可以用同样的方式来输出——也就是说，只需换掉 output 运算符右方的值即可。例如：

```
cout << "3 + 4 = ";  
cout << 3 + 4;  
cout << '\n';
```

会产生如下输出结果：

```
3 + 4 = 7
```

我们在自己的应用程序中定义了新的 class 时，也应该为每一个 class 提供它们自己的 output 运算符（第 4 章会告诉你如何办到这件事情）。这么一来便可以让那些 class 的用户得以像面对内置类型一样地以相同方式输出对象内容。

如果嫌连续数行的输出语句太烦人，也可以将数段内容连成单一输出语句：

```
cout << '\n'  
    << "Hello, "  
    << user_name  
    << " ... and goodbye!\n";
```

最后，我们以 return 语句清楚地表示 main() 到此结束：

```
return 0;
```

return 是 C++ 中的关键字。此例中的 0 是紧接于 return 之后的表达式（expression），也就是此函数的返回值。先前我曾说过，main() 返回 0 即表示程序执行成功¹。

将所有程序片段组合在一起，便是我们的第一个完整的 C++ 程序：

```
#include <iostream>  
#include <string>  
using namespace std;    // 此行目前尚未解释……  
  
int main()  
{  
    string user_name;  
    cout << "Please enter your first name: ";  
    cin >> user_name;  
    cout << '\n'  
        << "Hello, "
```

¹ 如果没有在 main() 的末尾写下 return 语句，这一语句会被自动加上。本书各程序范例中，我将不再明确写出 return 语句。

```
        << user_name
        << " ... and goodbye!\n";

    return 0;
}
```

编译并执行后，上述程序代码会产生如下输出结果（输入部分以粗体字表示）：

```
Please enter your first name: anna
Hello, anna ... and goodbye!
```

整个程序中还有一行语句我尚未解释：

```
using namespace std;
```

让我们瞧瞧，如果试着这样解释会不会吓到你（此刻，我建议你深吸一口气）。using 和 namespace 都是 C++ 中的关键字。std 是标准库所驻之命名空间（namespace）的名称。标准库所提供的任何事物（诸如 string class 以及 cout、cin 这两个 iostream 类对象）都被封装在命名空间 std 内。当然，或许你接下来会问，什么是命名空间？

所谓命名空间（namespace）是一种将库名称封装起来的方法。通过这种方法，可以避免和应用程序发生命名冲突的问题（所谓命名冲突是指在应用程序内两个不同的实体〔entity〕具有相同名称，导致程序无法区分两者。命名冲突发生时，程序必须等到该命名冲突获得解析〔resolve〕之后，才得以继续执行）。命名空间像是在众多名称的可见范围之间竖起的一道道围墙。

若要在程序中使用 string class 以及 cin、cout 这两个 iostream 类对象，我们不仅需要包含 <string> 及 <iostream> 头文件，还得让命名空间 std 内的名称曝光。而所谓的 using directive：

```
using namespace std;
```

便是让命名空间中的名称曝光的最简单方法。（[LIPPMAN98]的 8.5 节和[STROUSTRUP97]的 8.2 节有命名空间〔namespace〕的更多相关信息。）

练习 1.1

将先前介绍的 main() 程序依样画葫芦地输入。你可以直接输入程序代码，或是从网络上下载。本书前言已经告诉你如何获得书中范例程序的源程序以及习题解答。试着在你的系统上编译并执行这个程序。

练习 1.2

将 string 头文件注释掉：

```
// #include <string>
```

重新编译这个程序，看看会发生什么事。然后取消对 `string` 头文件的注释，再将下一行注释：

```
// using namespace std;
```

又会发生什么事情？

练习 1.3

将函数名 `main()` 改为 `my_main()`，然后重新编译。有什么结果？

练习 1.4

试着扩充这个程序的内容：(1) 要求用户同时输入名字（`first name`）和姓氏（`last name`），(2) 修改输出结果，同时打印姓氏和名字。

1.2 对象的定义与初始化

Defining and Initializing a Data Object

现在，我们的程序引起了用户的注意。让我们出个题目来考考他。我要显示某数列中的两个数字，然后要求用户回答下一个数字是什么。例如：

```
The values 2, 3 for two consecutive
elements of a numerical sequence.
What is the next value?
```

这两个数字事实上是“斐波那契数列（`Fibonacci sequence`）”中的第三和第四个元素。斐波那契数列的前几个值分别是：1, 1, 2, 3, 5, 8, 13…。斐波那契数列的开头两个数设定为 1，接下来的每个数值都是前两个数值的总和。（第 2 章会示范一个计算斐波那契数列的函数。）

如果用户输入 5，我们就打印出信息，恭喜他答对，并询问他是否愿意试试另一个数列。如果用户输入不正确的值，我们就询问他是否愿意再试一次。

为了提升程序的趣味性，我们将用户答对的次数除以其回答总次数，以此作为评价标准。

这样一来，我们的程序至少需要五个对象：一个 `string` 对象用来记录用户的名字，三个整数对象分别储存用户回答的数值、用户回答的次数，以及用户答对的次数；此外还需要一个浮点数，记录用户得到的评分。

为了定义对象，我们必须为它命名，并赋予它数据类型。对象名称可以是任何字母、数字、下划线（`underscore`）的组合。大小写字母是有所区分的，`user_name`、`User_name`、`uSeR_nAmE`、`user_Name` 所代表的对象各不相同。

对象名称不能以数字开头。举例来说，`1_name` 是不合法的名称，`name_1` 则合法。当然，任何命名都不能和程序语言本身的关键字完全一致。例如 `delete` 是语言关键字，就不可以用于程序内的命名。（这也正是 `string` class 采用 `erase()` 而非 `delete()` 来表示“删去一个字符”的原因。）

每个对象都属于某个特定的数据类型。对象名称如果设计得好，可以让我们直接联想到该对象的属性。数据类型决定了对象所能持有的数值范围，同时也决定了对象应该占用多少内存空间。

前一节中我们已经看过 `user_name` 的定义。新程序中我们再一次使用相同的定义：

```
#include <string>
string user_name;
```

所谓 `class`，便是程序员自行定义的数据类型。除此之外，C++还提供了一组内置的数据类型，包括布尔值（`Boolean`）、整数（`integer`）、浮点数（`floating point`）、字符（`character`）。每一个内置数据类型都有一个相应的关键字，用于指定该类型。例如，为了储存用户输入的值，我们定义一个整数对象：

```
int usr_val;
```

`int` 是 C++ 关键字，此处用来指示 `usr_val` 是个整数对象。用户的“回答次数”以及“总共答对次数”也都是整数，唯一差别是，我们希望为这两个对象设定初值 0。下面这两行可以办到：

```
int num_tries = 0;
int num_right = 0;
```

在单一声明语句中一并定义多个对象，其间以逗号分隔，也可以：

```
int num_tries = 0, num_right = 0;
```

一般来说，将每个对象初始化，是个好主意——即使初值只用来表示该对象尚未具有真正有意义的值。我之所以不为 `usr_val` 设置初值，是因为其值必须直接根据用户的输入加以设定，然后程序才能使用。

另外还有一种不同的初始化语法，称为“构造函数语法（`constructor syntax`）”：

```
int num_tries(0);
```

我知道你心中有疑问：为什么需要两种不同的初始化语法呢？为什么直到此刻才提起呢？唔，让我们看看下面这个解释能否回答其中一个问题，甚至同时回答这两个问题。

“用 `assignment` 运算符（`=`）进行初始化”这个操作系沿袭自 C 语言。如果对象属于内置类型，或者对象可以单一值加以初始化，这种方式就没有问题。例如以下的 `string` `class`：

```
string sequence_name = "Fibonacci";
```

但是如果对象需要多个初值，这种方式就没有办法完成任务了。以标准库中的复数（`complex number`）类为例，它就需要两个初值，一为实部，一为虚部。于是便引入了用来处理“多值初始化”的构造函数初始化语法（`constructor initialization syntax`）：

```
#include <complex>
complex<double> purei(0, 7);
```

出现于 `complex` 之后的尖括号，表示 `complex` 是一个 `template class`（模板类）。本书对于 `template class` 另有更详尽的讨论。`template class` 允许我们在“不必指明 `data member` 类型”的情况下定义 `class`。

举个例子，复数类内含两个 `member data object`。其一表示复数的实数部分，其二表示虚数部分。两者都需要以浮点数来表现，但我们应该采用哪种浮点数类型呢？C++ 支持三种浮点数类型，分别是以关键字 `float` 表示的单精度（single precision）浮点数，以关键字 `double` 表示的双精度（double precision）浮点数，以及连续两个关键字 `long double` 表示的长双精度（extended precision）浮点数。

`template class` 机制使程序员得以直到使用 `template class` 时才决定真正的数据类型。程序员可以先插入一个代号，稍后才绑定至实际的数据类型。上例便是将 `complex` 类的成员绑定至 `double` 类型。

我知道，这些说明带给你的只怕是疑问多过回答。然而，这是因为，当“内置数据类型”与“程序员自行定义的 `class` 类型”具备不同的初始化语法时，我们无法编写出一个 `template` 使它同时支持“内置类型”与“`class` 类型”。让语法统一，可以简化 `template` 的设计。不幸的是，解释这些语法似乎使事情益发复杂！

用户获得的评分可能是某个比值，所以我们必须以浮点数来表示。我以 `double` 类型来定义：

```
double usr_score = 0.0;
```

当我们询问“再试一次？”以及“你是否愿意回答其他类型的数列问题？”时，我们还必须将用户的回答（`yes` 或 `no`）记录下来。这个时候使用字符（`char`）对象就绰绰有余了：

```
char usr_more;
cout << "Try another sequence? Y/N? ";
cin >> usr_more;
```

关键字 `char` 表示字符（character）类型。单引号括住的字符代表所谓的字符常量，例如 `'a'`、`'7'`、`';`。此外还有一些特别的内置字符常量（有时也称为“转义字符 [escape sequence]”），例如：

```
'\n'   换行符 (newline)
'\t'   制表符 (tab)
'\0'   null
'\''   单引号 (single quote)
'\''   双引号 (double quote)
'\\'   反斜线 (backslash)
```

举个例子，我们想要在打印用户姓名之前，先换行并跳过一个制表符的（tab）距离，下面这行就可以办到：

```
cout << '\n' << '\t' << user_name;
```

另一种写法是将两个不同的字符合并为一个字符串：

```
cout << "\n\t" << user_name;
```

我们常常会在字符串常量中使用这些特殊字符。例如，在 Windows 操作系统下以字符串常量表示文件路径时，必须用“转义字符（escape sequence）”来表示反斜线字符：

```
"F: \\essential\\programs\\chapter1\\chl_main.cpp";
```

由于反斜线字符已用作转义字符的起头字符，因此连续两个反斜线即表示一个真正的反斜线字符。

C++提供了内置的 Boolean 类型，用以表示真假值（true/false）。我们的程序中可以定义 Boolean 对象来控制是否要显示下一组数列：

```
bool go_for_it = true;
```

Boolean 对象系由关键字 bool 指出。其值可为 true 或 false（两者都是常量）。

截至目前我们所定义出来的对象，其值都会在程序执行过程中改变。例如，go_for_it 最后会被设置成 false，用户每次猜完数字之后，usr_score 的值也可能更改。

但有时候我们需要一些用来表示常量的对象：比如用户最多可猜多少次，或者像圆周率这类永恒不变的值。这种对象的内容在程序执行过程中不应该有所变动。我们应当如何避免无意间更改此类对象的值呢？C++的 const 关键字可以派上用场：

```
const int max_tries = 3;  
const double pi = 3.14159;
```

被定义为 const 的对象，在获得初值之后，无法再有任何变动。如果你企图为 const 对象指定新值，会产生编译期错误。例如：

```
max_tries = 42;    // 错误：这是个 const 对象
```

1.3 撰写表达式

Writing Expressions

内置数据类型都可运用这样一组运算符，其中包括算术运算符、关系运算符、逻辑运算符、复合赋值（compound assignment）运算符。算术运算符中，除了“整数除法”以及“余数运算”外，其他并无出奇之处：

```
// 算术运算符 (Arithmetic Operator)  
+      加法运算      a + b  
-      减法运算      a - b  
*      乘法运算      a * b
```

/	除法运算	a / b
%	取余数	a % b

两个整数相除会产生另一个整数（商）。小数点之后的部分会被完全舍弃；也就是说，并没有四舍五入。如果想要取得除法运算的余数部分，可以使用%运算符：

5 / 3 值为 1	5 % 3 值为 2
5 / 4 值为 1	5 % 4 值为 1
5 / 5 值为 1	5 % 5 值为 0

嗯，什么时机之下，我们会运用“余数运算”呢？假设我们希望打印的数据每行不超过八个字符串；尚未满八个字符串时，就在字符串之后打印一个空格。如果已满八个字符串，就在字符串之后输出换行符。以下便是实现方法：

```
const int line_size = 8;
int cnt = 1;

// 以下语句将被我执行多次，每次 a_string 的内容
// 都不相同；每次执行完后，cnt 的值都会加 1。
cout << a_string
      << ( cnt % line_size ? ' ' : '\n');
```

其中紧接在 output 运算符（<<）之后，以括号括住的表达式（expression），是所谓的条件运算符（conditional operator）。如果余数运算的结果为零，则条件运算的结果为'\n'；如果余数运算的结果非零，则条件运算的结果为' '。让我们看看这到底是什么意思。

下面这个表达式：

```
cnt % line_size
```

在 cnt 恰为 line_size 的整数倍时，运算结果为零，反之则非零。而条件运算符的一般使用形式如下：

```
expr
? 如果 expr 为 true，就执行这里
: 如果 expr 为 false，则执行这里
```

如果 expr 的运算结果为 true，那么紧接在 '?' 之后的表达式会被执行。如果 expr 的运算结果为 false，那么 ':' 之后的表达式会被执行。在我们的程序中会导致喂给 output 运算符一个' '或是'\n'。

条件表达式的值如果为零，会被视为 false，其他非零值则一律被视为 true。本例中的 cnt 若非 8 的整数倍，条件表达式的值便不是零，于是条件运算符中的 '?' 之后的部分就会被求值。

复合赋值（compound assignment）运算符是一种简便的表达方法。我们在对象身上使用某个运算符，然后将结果重新赋值给该对象时，可能会这样写：


```
cnt = cnt + 2;
```

但是C++程序员通常会写成下面这样:

```
cnt +=2 ;    // cnt 原值加 2
```

复合赋值运算符可以和每个算术运算符结合, 形成+=、-=、*=、/=和%=。

欲使对象值递增或递减, C++程序员会使用递增 (increment) 运算符和递减 (decrement) 运算符:

```
cnt++;    // cnt 的值递增 1
```

```
cnt--;    // cnt 的值递减 1
```

递增运算符和递减运算符都有前置 (prefix) 和后置 (postfix) 两种形式。前置形式中, 原值先递增 (或递减), 之后才被拿来使用:

```
int tries = 0;
cout << "Are you ready for try #"
     << ++tries << "?\n";
```

上例中的 tries 在被打印之前就已进行了递增运算。至于后置形式写法, 对象原值会先供给表达式进行运算, 然后才递增 (或递减):

```
int tries = 1;
cout << "Are you ready for try #"
     << tries++ << "?\n";
```

上例中的 tries 在被打印之后才进行递增运算。以上两例的打印结果皆为 1。

任何一个关系运算符 (relational operator) 的求值结果不是 true 就是 false。关系运算符包括以下六个:

==	相等	a == b
!=	不等	a != b
<	小于	a < b
>	大于	a > b
<=	小于等于	a <= b
>=	大于等于	a >= b

我们可利用相等 (equality) 运算符来检验用户的回答:

```
bool usr_more = true;
char usr_rsp;

// 询问用户是否愿意继续下一个问题
// 将用户的回答读入 usr_rsp 之中
if ( usr_rsp == 'N' )
    usr_more = false;
```

if 语句之后的表达式运算结果为 true 时，条件成立，于是紧接其后的语句便会被执行。此例中如果 usr_rsp 等于 'N'，则 usr_more 会被设置为 false。反之，如果 usr_rsp 不等于 'N'，那就什么事也不会发生。不等运算符（inequality operator）的逻辑恰恰相反，例如：

```
if ( usr_rsp != 'Y' )
    usr_more = false;
```

麻烦的是，程序只检验 usr_rsp 的值是否为 'N'，而用户却可能输入小写的 'n'。因此我们必须能够辨识两者。解决方法之一就是加上 else 子句：

```
if ( usr_rsp == 'N' )
    usr_more = false;
else
    if ( usr_rsp == 'n' )
        usr_more = false;
```

如果 usr_rsp 的值为 'N'，usr_more 将被设置为 false，并结束整个 if 语句。如果其值不等于 'N'，那么便开始对接下来的 else 子句求值。换句话说，当 usr_rsp 等于 'n' 时，usr_more 也会被置成 false。如果两个条件都不符合，则 usr_more 不会被赋值。

新手常犯的错误便是将赋值（assignment）运算符误当作相等（equality）使用，例如：

```
// 呃，这会造成将常量字符 'N' 赋值给 usr_rsp
// 而整个表达式的运算结果为 true
if ( usr_rsp = 'N' )
    // ...
```

OR 逻辑运算符（||）提供了上述问题的另一个解法。它让我们得以同时检验多个表达式的结果：

```
if ( usr_rsp == 'N' || usr_rsp == 'n' )
    usr_more = false;
```

只需左右两个表达式中的一个为 true，OR 逻辑运算符的求值结果便为 true。左侧表达式会先被求值，如果其值为 true，剩下的另一个表达式就不需要再被求值（此所谓短路求值法）。本例中，只有当 usr_rsp 不等于 'N' 时，才会再检验其值是否为 'n'。

AND 逻辑运算符（&&）则在左右两个表达式的结果皆为 true 时，其求值结果方为 true。例如：

```
if ( password &&
    validate( password ) &&
    ( acct = retrieve_acct_info( password ) ) )
    // 处理账户 (account) 相关事务
```

最上面的一条表达式会先被求值。其结果若为 false，则 AND 运算符的求值结果即为 false，其余表达式不会（不需要）再被求值。本例中，当密码已设定，而且有效之后，账户的相关信息才会被取出。

如果有一个表达式的运算结果为 `false`，那么将 `NOT` 逻辑运算符用于其上，结果为 `true`。例如，我们可以把以下式子

```
if ( usr_more == false )
    cout << "Your score for this session is "
        << usr_score << " Bye!\n";
```

写成

```
if ( !usr_more ) ...
```

运算符的优先级

使用内置运算符时，你得明白一件事：如果同一个表达式中使用多个运算符，其求值（*evaluate*）顺序是由每一个运算符事先定义的优先级来决定。例如 $5+2*10$ 的运算结果是 25 而非 70，这是因为乘法的优先级比加法高，因此 2 要先和 10 相乘，然后再加上 5。

如果想要改变内置的运算符优先级，可利用小括号。例如 $(5+2)*10$ 的运算结果便是 70。

我将截至目前介绍过的运算符优先级简列于下。位置在上者的优先级高于位置在下者。同一行的各种运算符具有相同的优先级，其求值次序取决于它在该表达式中的位置（由左至右）。

- 逻辑运算符 NOT
- 算术运算符（`*`，`/`，`%`）
- 算术运算符（`+`，`-`）
- 关系运算符（`<`，`>`，`<=`，`>=`）
- 关系运算符（`==`，`!=`）
- 逻辑运算符 AND
- 逻辑运算符 OR
- 赋值（assignment）运算符

举个例子，当我们想判断 `ival` 是否为偶数时，可能会这么写：

```
! ival % 2    // 不正确
```

我们的想法是利用余数运算符（`%`）来检验其结果。如果 `ival` 是偶数，则余数运算的结果为零，进行逻辑运算 `NOT` 之后，结果为 `true`。如果余数运算的结果不为零，进行逻辑运算 `NOT` 之后，结果便为 `false`。

不幸的是，上述表达式的结果和我们的想象大相径庭。除非 `ival` 等于零，否则表达式结果总是 `false`。

为什么？因为逻辑运算 `NOT` 具备较高的优先级，将最先被求值。因此它先被作用于 `ival` 之上：如果 `ival` 不为零，则运算结果为 `false`，反之则为 `true`。这个结果再成为余数运算的左操作数。而你知道，`false` 在算术表达式中被视为 0，`true` 被视为 1，所以，根据 C++ 默认的运算优先级，除非 `ival` 的值为零，否则上述表达式便成了 $0\%2$ 。

虽然这样的结果并不是我们想要的，但也不能算是错误，嗯，至少不能算是语言上的错误。只能说是我们的程序逻辑的一种不正确的表达方式。编译器不可能知道你的程序的逻辑。运算优先级是 C++ 编程之所以复杂的原因之一。如果希望正确计算以上表达式，我们应该明确地加上小括号，来实现我们希望的运算优先级：

```
! ( ival % 2 )    // 正确的写法
```

要避免此类问题，你必须坐下来好好深入地熟悉 C++ 运算符优先级。我并不打算在这里介绍所有运算符，以及所有的运算符优先级，以上介绍对于正在起步的初学者而言，应已足够。如果想获得更完整的说明，请参考[LIPPMAN98]的第 4 章或[STROUSTRUP97]的第 6 章。

1.4 条件语句和循环语句

Writing Conditional and Loop Statements

基本上，从 `main()` 的第一行语句（statement）开始，程序里的每行语句都只会被依序执行一次。我们已经在前面的小节中对 `if` 语句做了惊鸿一瞥。`if` 语句让我们依据某个表达式的结果（视为真/假值）来决定是否执行一条或多条连续的语句。可有可无的 `else` 子句，更可让我们连续检验多个测试条件。至于循环（loop）语句，可以让我们根据某个表达式结果（视为真/伪条件），重复执行单一或连续多条语句。下面以伪码（pseudocode）所表示的程序中，使用了两组循环语句（#1, #2）、一组 `if` 语句（#5）、一组 `if-else` 语句（#3），以及一组称为 `switch` 语句的条件语句（#4）。

```
// 伪码 (pseudocode): 程序逻辑的一般化表示
while 用户想要猜测某个数列时
{ #1
    显示该数列
    while 用户所猜的答案并不正确 and
        用户想要再猜一次
    { #2
        读取用户所猜的答案
        递增 number_of_tries
        if 答案正确
        { #3
            递增 correct_guess
            将 got_it 的值设为 true
        } else {
            用户答错了，在此对他表示遗憾，并
            根据用户已猜过的总数，产生不同的
            回应结果。 // #4
            询问用户是否愿意再试一次
            读取用户的意愿
            if 用户响应 no // #5
```

将 go_for_it 的值设为 false

```
    }
}
}
```

条件语句

if 语句中的条件表达式(condition expression)必须写在括号内。如果此表达式的运算结果为 true, 那么紧接在 if 之后的那一条语句便会被执行。

```
// #5
if ( usr_rsp == 'N' || usr_rsp == 'n' )
    go_for_it = false;
```

如果想执行多条语句, 那么必须在 if 之后以大括号将这些语句括住(这样便称为一个语句块):

```
// #3
if ( usr_guess == next_elem )
{ // 语句块由此开始
    num_right++;
    got_it = true;
} // 语句块在此结束
```

初学者常见的错误是忘了加上语句块:

```
// 呃: 忘了加上语句块
// 两条语句之中, 只有 num_cor++是在 if 语句的控制范围内。
// 至于 got_it = true;这一行, 不管条件是否成立都会被执行。
```

```
if ( usr_guess == next_elem )
    num_cor++;
    got_it = true;
```

got_it 之前的缩进反映出了程序员的意图。但是这并不会改变程序本身的行为。是的, num_cor 的递增与否与 if 语句相关, 而且只有在用户猜测的值(usr_guess)和下一元素之值(next_elem)相等时, 它才会被执行。但是接下来的 got_it 语句却和 if 语句丝毫不关, 因为我们忘了将这两条语句置于同一语句块中。因此, 本例中不论用户所猜的值是什么, got_it 一定会被设置为 true。

if 语句也可以配合 else 子句来使用。else 子句用以表示一旦 if 语句的测试条件不成立, 我们所希望执行的单行语句或语句块。

```
if (usr_guess == next_elem)
{
    // 用户猜对了
}
else
{
    // 用户猜错了
}
```

使用 `else` 子句的第二种方式，便是将它和两条（或更多）`if` 语句结合。举例来说，如果用户猜错了，我们希望输出的响应能够依据用户所猜过的总次数而有所变化，这时候我们可以编写连续三条独立的 `if` 语句来加以检验：

```
if ( num_tries == 1 )
    cout << "Oops! Nice guess but not quite it.\n";

if ( num_tries == 2 )
    cout << "Hmm. Sorry. Wrong a second time.\n";

if ( num_tries == 3 )
    cout << "Ah, this is harder than it looks, isn't it?\n";
```

三个测试条件中仅有一个会成立。如果其中一条 `if` 语句为 `true`，其他两者必为 `false`。因此我们可以结合一连串的 `else-if` 子句，来反映这些 `if` 语句间的关系，也就是：

```
if ( num_tries == 1 )
    cout << "Oops! Nice guess but not quite it.\n";
else
if ( num_tries == 2 )
    cout << "Hmm. Sorry. Wrong a second time.\n";
else
if ( num_tries == 3 )
    cout << "Ah, this is harder than it looks, isn't it?\n";
else
    cout << "It must be getting pretty frustrating by now!\n";
```

第一条 `if` 语句会先被求值。如果其值为 `true`，紧接在后的语句便会被执行，接下来的所有 `else-if` 子句都不会被求值。但如果第一条 `if` 语句的求值结果为 `false`，便会接着对下一条 `if` 语句求值，直到其中有某一个条件成立为止。如果 `num_tries` 的值大于 3，也就是说所有条件都不成立，那么最末的 `else` 子句会被执行。

嵌套的（`nested`）`if-else` 子句有个很容易令人困惑的地方，那就是要正确组织其逻辑其实是比较难的一件事。比如，我们想要利用 `if-else` 语句将程序的执行分为两种情形：(1) 用户猜对，(2) 用户猜错。以下第一种写法并不会如我们所预期的方式工作：

```
if ( usr_guess == next_elem )
{
    // 用户猜对了
}
else
if ( num_tries == 1 )
    // 输出适当的响应结果
else
if ( num_tries == 2 )
    // 输出适当的响应结果
else
```

```
if ( num_tries == 3 )
    // 输出适当的响应结果
else
    // 输出适当的响应结果

// 现在, 询问用户是否愿意再猜一次。
// 但只有在用户猜错时我们才应该这么做。
// 呃, 我们应该把程序代码放在哪儿呢?
```

每条 else-if 子句都无意识地形成了二选一的命运, 于是我们找不到地方安置处理用户猜错时的程序代码。以下是正确的组织方式:

```
if ( usr_guess == next_elem )
{
    // 用户猜对了
}
else
{
    // 用户猜错了
    if ( num_tries == 1 )
        // ...
    else
    if ( num_tries == 2 )
        // ...
    else
    if ( num_tries == 3 )
        // ...
    else // ...

    cout << "Want to try again? (Y/N)";
    char usr_rsp;
    cin >> usr_rsp;

    if ( usr_rsp == 'N' || urs_rsp == 'n' )
        go_for_it = false;
}
```

如果测试条件值属于整数类型, 我们还可以改用 switch 语句来取代一大堆的 if-else-if 子句:

```
// 等同于上述的 if-else-if 子句
switch ( num_tries )
{
    case 1:
        cout << "Oops! Nice guess but not quite it.\n";
        break;

    case 2:
        cout << "Hmm. Sorry. Wrong a second time.\n";
        break;
```



```
case 3:
    cout << "Ah, this is harder than it looks, isn't it?\n";
    break;

default:
    cout << "It must be getting pretty frustrating by now!\n";
    break;
}
```

关键字 `switch` 之后紧接着一个由小括号括住的表达式（是的，对象名称也可视为表达式），该表达式的值必须是整数形式。关键字 `switch` 之后是一组 `case` 标签，每一个标签之后都指定有一个常量表达式。在 `switch` 之后的表达式值被计算出来后，便依次和每个 `case` 标签的表达式值相比较。如果找到相符的 `case` 标签，便执行该 `case` 标签之后的语句。如果找不到吻合者，但有 `default` 标签，便执行 `default` 标签之后的语句。如果 `default` 标签也没有，就不执行任何操作。

为什么我在每个 `case` 标签的最末加上 `break` 语句呢？要知道，每个 `case` 标签的表达式值都会依次和 `switch` 表达式值相比较，不吻合者便依次跳过。若某个 `case` 标签的表达式值吻合，便开始执行该 `case` 标签之后的语句——这个操作会一直执行到 `switch` 语句的最下面。如果我们没有加上 `break` 语句，而 `num_tries` 的值为 2，那么程序的输出结果会是：

```
// 如果 num_tries 值为 2，而我们又忘了加上 break 语句，输出结果如下
Hmm. Sorry. Wrong a second time.
Ah, this is harder than it looks, isn't it?
It must be getting pretty frustrating by now!
```

是的，当某个标签和 `switch` 的表达式值吻合时，该 `case` 标签之后的所有 `case` 标签也都会被执行，除非我们明确使用 `break` 来结束执行。这也正是 `break` 语句的用途。或许你心中产生了疑问，为什么要把 `switch` 语句设计成这样呢？下面的例子可以说明这种“向下穿越”的行为模式的合理性：

```
switch( next_char )
{
    case 'a': case 'A':
    case 'e': case 'E':
    case 'i': case 'I':
    case 'o': case 'O':
    case 'u': case 'U':
        ++vowel_cnt;
        break;
    // ...
}
```

循环语句

只要条件表达式不断成立（亦即其运算结果为 `true`），循环语句便会不断地执行单一语句或整个语句块。我们的程序需要用到两个循环语句，其中一个被嵌套在另一个循环之中：

```
while 用户想要猜数列的下一个数
{
    显示数列
    while 用户猜的答案并不正确 and
        用户想要再猜一次
}
```

C++中的 `while` 循环可以满足我们的需求：

```
bool next_seq = true;    // 显示下一组数列
bool go_for_it = true;   // 用户想再猜一次
bool got_it = false;     // 用户是否猜对
int  num_tries = 0;       // 用户猜过的总次数
int  num_right = 0;       // 用户答对的总次数

while ( next_seq == true )
{
    // 为用户显示数列
    while (( got_it == false) &&
           ( go_for_it == true ))
    {
        int usr_guess;
        cin >> usr_guess;
        num_tries++;
        if ( usr_guess == next_elem )
        {
            got_it = true;
            num_right++;
        }
        else
        {
            // 用户猜错了
            // 告诉用户答案是错的
            // 询问用户是否愿意再试一次
            if ( usr_rsp == 'N' || usr_rsp == 'n' )
                go_for_it = false;
        }
    } // 内层的 while 循环结束

    cout << "Want to try another sequence? (Y/N)";
    char try_again;
    cin >> try_again;

    if ( try_again == 'N' || try_again == 'n' )
        next_seq = false;
```

```
    } // while( next_seq == true ) 结束
```

while 循环即将开始之际，会先对括号内的条件表达式求值。如果其值为 true，则紧接在 while 循环之后的语句或语句块便会被执行。执行完毕之后，条件表达式会被重新求值。这种求值/执行的工作模式不断循环，直到条件表达式的值变成 false。通常，语句块在某种状态下会将该条件表达式的值设置为 false。如果条件表达式的值永远不为 false，我们便陷入了一个无穷循环之中——这样的程序逻辑当然不正确。

以上程序的外层 while 循环会一直执行到用户希望结束为止。

```
bool next_seq = true;
while ( next_seq == true )
{
    // ...
    if ( try_again == 'N' || try_again == 'n' )
        next_seq = false;
}
```

如果 next_seq 初值为 false，后面的语句块都不会被执行。至于内层的 while 循环则让用户得以连续猜好几次。

如果在执行循环内的语句时遇上 break 语句，循环便会结束。以下列程序片段为例，while 循环会一直执行到 tries_cnt 和 max_tries 相等为止。一旦用户猜对，程序便以 break 语句来结束循环：

```
int max_tries = 3;
int tries_cnt = 0;
while ( tries_cnt < max_tries )
{
    // 读取用户的答案
    if ( usr_guess == next_elem )
        break; // 结束循环

    tries_cnt++;
    // 其他程序代码
}
```

我们也可以利用 continue 语句来遽然终止循环的当前迭代（current iteration）。例如以下程序片段，所有长度小于四个字符的单词都会被舍弃掉：

```
string word;
const int min_size = 4;
while ( cin >> word )
{
    if ( word.size() < min_size )
        // 结束此次迭代
        continue;
}
```

```
// 程序执行到此处，则用户所输入的单词长度
// 必然大于或等于 min_size 个字符……
process_text(word);
}
```

倘若 word 的长度小于 min_size，便执行 continue 语句，于是结束循环的当前迭代，也就是说，while 循环中的剩余部分（本例为 process_text()）便不会被执行。循环会重新来过，而条件表达式会被再一次求值——读取另一个字符串并存入 word 之中。如果 word 的长度大于或等于 min_size，整个 while 循环内容便都会被执行。在此工作模式下，所有长度小于四个字符的单词都会被舍弃掉。

1.5 如何运用 Array 和 Vector

How to Use Arrays and Vectors

以下是六种数列的前八个元素值：

```
Fibonacci: 1, 1, 2, 3, 5, 8, 13, 21
Lucas:      1, 3, 4, 7, 11, 18, 29, 47
Pell:       1, 2, 5, 12, 29, 70, 169, 408
Triangular: 1, 3, 6, 10, 15, 21, 28, 36
Square:     1, 4, 9, 16, 25, 36, 49, 64
Pentagonal: 1, 5, 12, 22, 35, 51, 70, 92
```

我们的程序必须能够显示数列中的任意两个元素值，让用户猜测下一个元素值是什么。如果用户猜对了，并且愿意继续下去，程序便接着显示第二组、第三组……元素值。这要如何办到呢？

如果接着出现的每组元素都出自同一数列，那么一旦用户找出其中一组答案，他就能够找出所有答案。这就丧失了趣味性。所以，我们应该在程序主循环的每次迭代中，挑选不同的数列。

现在，我们要显示最多六组元素对（element pair）：每一组来自不同的数列。我们希望在显示每组元素的同时，不必知道正在显示的是哪一种数列。每次迭代都必须存取三个数：“元素对”中的两个元素值，以及数列中出现的第三个元素值。

本节所讨论的问题，解决之道就是使用可存放连续整数值的容器（container）类型。这种类型不仅允许我们以名称（name）取用容器中的元素，也允许我们以容器中的位置来取用元素。我打算在容器内放入 18 个数值，分为六组。每一组的前两个数值用于显示，第三个数值表示数列中的下一元素值。在循环的每次迭代过程中，我们令索引（index）每次增加 3，这样就可以依次走访六组数据。

C++允许我们以内置的 array（数组）类型或标准库提供的 vector 类来定义容器。一般而言，我建议使用 vector 甚于 array。不过，大量现存的程序代码都使用 array。因此，了解如何善用这两种方式，便相当重要。

要定义 array，我们必须指定 array 的元素类型，还得给予 array 一个名称，并指定其尺度大小——亦即 array 所能储存的元素个数。array 的大小必须是个常量表达式（constant expression），也就是一个不需要在运行时求值的表达式。以下程序代码是个例子，其中声明了 `pell_seq` 是个 array，内含 18 个整数元素。

```
const int seq_size = 18;
int pell_seq[ seq_size ];
```

至于定义 vector object，我们首先必须包含 `vector` 头文件。`vector` 是个 class template，所以我们必须在类名之后的尖括号内指定其元素类型，其大小则写在小括号中；此处所给予的大小并不一定得是个常量表达式。下列程序代码将 `pell_seq` 定义为一个 vector object，可储存 18 个 int 元素，每个元素的初值为 0。

```
#include <vector>
vector<int> pell_seq( seq_size );
```

无论 array 或 vector，我们都可以指定容器中的某个位置，进而访问该位置上的元素。索引操作（indexing）系通过下标运算符（`[]`）达成。这里必须注意的小技巧是，容器的第一个元素位置为 0 而非 1。因此，容器的最后一个元素必须以容器的大小减一加以索引。以 `pell_seq` 为例，正确的索引值是 0~17，而非 1~18（此类错误极为常见，因此这种错误有个声名狼藉的名称，叫做 `off-by-one`）。举个例子，要指定 Pell 数列的前两个元素值，可以这么写：

```
pell_seq[ 0 ] = 1; // 指定第一元素值为 1
pell_seq[ 1 ] = 2; // 指定第二元素值为 2
```

现在我们来计算 Pell 数列中接下来的十个元素。要依次迭代 vector 或 array 中的多个元素时，我们通常会使用 C++ 中的另一种重要的循环语句，for 循环。例如：

```
for ( int ix = 2; ix < seq_size; ++ix )
    pell_seq[ ix ] = pell_seq[ ix-2 ] + 2*pell_seq[ ix-1 ];
```

for 循环包含以下几个组成部分：

```
for ( init-statement ; condition ; expression )
    statement
```

其中的 `init-statement` 会在循环开始执行前被执行一次。在我们的例子中，`ix` 在循环开始之前，被设置初值为 2。

`condition` 用于循环控制，其值会在每次循环迭代之前被计算出来。如果 `condition` 为 `true`，`statement` 便会被执行。`statement` 可以是单一语句，也可以是个语句块。如果 `condition` 第一次求值即为 `false`，那么 `statement` 一次也不会执行。在我们的例子中，`condition` 用来检验 `ix` 的值是否小于 `seq_size`。

`expression` 会在循环每次迭代结束之后被求值。通常它用来更改两种对象的值。一个是在 `init-statement` 中被初始化的对象，另一个是在 `condition` 中被检验的对象。如果 `condition` 第一次求值即为 `false`，那么 `expression` 不会被执行。在我们的例子中，每次循环迭代结束之后 `ix` 的值便递增 1。

如果想打印出每一个元素值，我们可以对整个集合进行迭代 (*iterate*):

```
cout << "The first " << seq_size
      << " elements of the Pell Series:\n\t";

for ( int ix = 0; ix < seq_size; ++ix )
    cout << pell_seq[ ix ] << ' ';

cout << '\n';
```

如果我们愿意，也可以在 `init-statement` 或 `expression` 甚至 `condition` 处（较罕见）留空，不写任何东西。例如，我们可以将上述的 `for` 循环改为

```
int ix = 0;
// ...

for ( ; ix < seq_size; ++ix )
    // ...
```

其中的分号是必要的，因为必须利用它来表示 `init-statement` 留空。

我们的容器储存了六个数列中的每一个数列的第二、第三、第四个元素。我们应该如何将适当的值填入此容器中呢？如果是 `array`，我们可以指定初始化列表 (`initialization list`)，借由逗号分隔每一个要指定的值，这些值便成为 `array` 的全部或部分元素：

```
int elem_seq[ seq_size ] = {
    1, 2, 3,    // Fibonacci
    3, 4, 7,    // Lucas
    2, 5, 12,   // Pell
    3, 6, 10,   // Triangular
    4, 9, 16,   // Square
    5, 12, 22   // Pentagonal
};
```

初始化列表内的元素个数，不能超过 `array` 的大小。如果前者的元素数量小于 `array` 的大小，其余的元素值会被初始化为 0。如果我们愿意，可以让编译器根据初值的数量，自行计算出 `array` 的大小：

```
// 编译器会算出此 array 包含了 18 个元素
int elem_seq[] = {
    1, 2, 3, 3, 4, 7, 2, 5, 12,
    3, 6, 10, 4, 9, 16, 5, 12, 22
};
```

`vector` 不支持上述这种初始化列表。有个冗长的写法可以为每个元素指定其值：

```
vector<int> elem_seq( seq_size );
elem_seq[ 0 ] = 1;
elem_seq[ 1 ] = 2;
// ...
elem_seq[ 17 ] = 22;
```

另一个方法是利用一个已初始化的 `array` 作为该 `vector` 的初值：

```
int elem_vals[ seq_size ] = {
    1, 2, 3, 3, 4, 7, 2, 5, 12,
    3, 6, 10, 4, 9, 16, 5, 12, 22
};

// 以 elem_vals 的值来初始化 elem_seq
vector<int> elem_seq( elem_vals, elem_vals+seq_size );
```

上例中我们传入两个值给 `elem_seq`。这两个值都是实际内存位置。它们标示出了“用以将 `vector` 初始化”的元素范围。本例中我们标示出了 `elem_vals` 内的 18 个元素，并将它们复制到 `elem_seq`。第 3 章会详细介绍这种工作方式。

现在，让我们看看如何使用 `elem_seq`。`array` 和 `vector` 之间存在一点差异，那就是 `vector` 知道自己大小是多少。之前我们以 `for` 循环迭代 `array` 的做法，如果应用于 `vector` 之上，情况将稍有不同：

```
// elem_seq.size() 会返回 elem_seq 这个 vector
// 所包含的元素个数
cout << " The first " << elem_seq.size()
      << " elements of the Pell Series:\n\t";

for ( int ix = 0; ix < elem_seq.size(); ++ix )
    cout << pell_seq[ ix ] << ' ';
```

下面以 `cur_tuple` 表示要显示的元素的索引值。首先将它初始化为 0。每次循环迭代中，我们将其值累加 3，使它能够索引到下一个数列的第一元素。

```
int cur_tuple = 0;

while ( next_seq == true &&
        cur_tuple < seq_size )
{
    cout << "The first two elements of the sequence are: "
          << elem_seq[ cur_tuple ] << ", "
          << elem_seq[ cur_tuple+1 ]
          << "\nWhat is the next element?";
```



```

// ...
if ( usr_guess == elem_seq[ cur_tuple+2 ] )
    // correct!

// ...
if ( usr_rsp == 'N' || usr_rsp == 'n' )
    next_seq = false;
else cur_tuple += 3;
}

```

将目前进行中的数列类别记录下来, 应该颇有用处。首先, 我们将每个数列的名称都用 `string` 储存起来:

```

const int max_seq = 6;
string seq_names[ max_seq ] = {
    "Fibonacci",
    "Lucas",
    "Pell",
    "Triangular",
    "Square",
    "Pentagonal"
};

```

然后我们可以像下面这样来运用 `seq_names`:

```

if ( usr_guess == elem_seq[ cur_tuple+2 ] )
{
    ++num_cor;
    cout << "Very good. Yes, "
        << elem_seq[ cur_tuple+2 ]
        << " is the next element in the "
        << seq_names[ cur_tuple/3 ] << "sequence.\n";
}

```

`cur_tuple/3` 这一表达式会依次产生 0, 1, 2, 3, 4, 5, 用来索引出一个字符串, 代表目前正在进行的猜数游戏中的数列类别。

1.6 指针带来弹性

Pointers Allow for Flexibility

前一节所显示的解法有两大缺点。第一, 其上限是六个数列; 如果用户猜完了这六个数列, 程序会无预期地结束。第二, 这个方法每次都以同样的顺序显示六组元素。如何才能够增加程序的弹性呢?

一种可能的解法便是同时维护六个 `vector`, 每个数列使用一个。每个 `vector` 储存某一数量的元素值。每一次循环迭代, 我们从不同的 `vector` 取出一组元素值。当第二次用到相同的 `vector` 时, 便以不同的索引值取出 `vector` 内的元素。这个方法可以解决上述缺点。

如同早先的解法一样，我们希望透明地访问不同的 `vector`。前一节系通过索引（而非名称）来访问每个元素，借此达到透明化的目的。每次循环迭代，我们将索引累加 3。如若不然，程序的执行结果就不会改变。

这一节我们将通过使用指针（`pointer`），舍弃以名称指定的方式，间接地访问每个 `vector`，借此达到透明化的目的。指针为程序引入了一层间接性。我们可以操作指针（代表某特定内存地址），而不再直接操作对象。在我们的程序中，我定义了一个可以对整数 `vector` 寻址的指针。每一次循环迭代，便更改指针值，使它定位到不同的 `vector`。随后的指针操作行为不需要更改。

在我们的程序中，指针主要做两件事情。它可以增加程序本身的弹性，但同时也增加了直接操作对象时所没有的复杂度。本节内容会令你相信这两点。

我们早已了解如何定义对象。以下语句将 `ival` 定义为一个 `int` 对象，并给予初值 1024：

```
int ival = 1024;
```

指针内含某特定类型对象的内存地址。当我们要定义某个特定类型的指针时，必须在类型名称之后加上*号：

```
int *pi; // pi 是个 int 类型对象的指针
```

`pi` 是个 `int` 类型对象的指针。我们应当如何为指针设定初值呢？如果以对象名称来执行求值操作，例如：

```
ival; // 计算 ival 的值
```

会得到 `ival` 所存的值。如果希望取得对象所在的内存地址而非对象的值，则应该使用取址运算符（`&`）：

```
&ival; // 计算 ival 所在的内存地址
```

下述写法可将 `pi` 的初值设置为 `ival` 所在的内存地址：

```
int *pi = &ival;
```

如果要访问一个由指针所指的对象，我们必须对该指针进行提领（`dereference`）操作——也就是取得“位于该指针所指内存地址上”的对象。在指针之前使用*号，便可以达到这个目的：

```
// 提领 pi，借以访问它所指向的对象
if ( *pi != 1024 ) // 读取 ival 的值
    *pi = 1024;    // 写值至 ival
```

指针的复杂度，如你所见，源于其令人困惑的语法。本例中可能令人感到复杂的地方，便是指针所具有的双重性质：既可以让我们操作指针包含的内存地址，也可以让我们操作指针所指的对象值。如果我们这么写：

```
pi; // 计算 pi 所持有的内存地址
```

此举形同操作“指针对象”本身。而如果写

```
*pi; // 求 ival 的值
```

等于是操作 `pi` 所指的对象。

指针的第二个可能令人感到复杂的地方是，指针可能并不指向任何对象。当我们写 `*pi` 时，这种写法可能会（也可能不会）使程序在运行时产生错误。如果 `pi` 定位到某个对象，则对 `pi` 进行提领（dereference）操作没有错误。但如果 `pi` 不指向任何对象，则提领 `pi` 会导致未知的执行结果。这意味着，在使用指针时，必须在提领之前先确定它的确指向某对象。该怎么做呢？

一个未指向任何对象的指针，其地址值为 0。有时候我们称之为 `null` 指针。任何指针都可以被初始化，或是令其值为 0。

```
// 初始化每个指针，使它们不指向任何对象
```

```
int *pi = 0;
double *pd = 0;
string *ps = 0;
```

为了防止对 `null` 指针进行提领操作，我们可以检验该指针所持有的地址是否为 0。例如：

```
if ( pi && *pi != 1024 )
    *pi = 1024;
```

以下这个表达式

```
if ( pi && ... )
```

只有在 `pi` 持有一个非零值时，其求值结果才会是 `true`。如果求值结果为 `false`，那么 AND 运算符就不会对其第二表达式求值。欲检验某指针是否为 `null`，我们通常使用逻辑运算符 NOT：

```
if ( !pi ) // 当 pi 值为 0 时，此表达式方为 true
```

以下便是我们的六个 `vector` 对象（代表六种数列）：

```
vector<int> fibonacci, lucas, pell, triangular, square, pentagonal;
```

当我们需要一个指针，指向一个“元素类型为 `int`”的 `vector` 时，该指针应该是什么模样呢？通常，指针具有以下形式：

```
type_of_object_pointed_to * name_of_pointer_object
```

由于我们所要的指针系用来指向 `vector<int>`，因此我把它命名为 `pv`，并给定初值 0：

```
vector<int> *pv = 0;
```

`pv` 可以依次指向每一个表示数列的 `vector`。当然，我们也可以明确地将数列的内存地址赋值给它：

```
pv = &fibonacci;
// ...
pv = &lucas;
```

但这种赋值方式会牺牲程序的透明性。另一种方案是将每个数列的内存地址存入某个 `vector` 中，这样我们就可以通过索引的方式，透明地访问这些数列：

```
const int seq_cnt = 6;

// 一个指针数组，大小为 seq_cnt，
// 每个指针都指向 vector<int>对象
vector<int> *seq_addrs[ seq_cnt ] = {
    &fibonacci, &lucas, &pell,
    &triangular, &square, &pentagonal
};
```

`seq_addrs` 是个 `array`，其元素类型为 `vector<int> *`。`seq_addrs[0]` 所持有的值是 `fibonacci` `vector` 的地址，`seq_addrs[1]` 的值是 `lucas` `vector` 的地址，以此类推。我们通过一个索引值而非其名称来访问各个 `vector`：

```
vector<int> *current_vec = 0;
// ...

for ( int ix = 0; ix < seq_cnt; ++ix )
{
    current_vec = seq_addrs[ ix ];
    // 所有要显示的元素都通过 current_vec 间接访问到
}
```

最后，剩下一个问题悬而未决。在这种实现方式下，程序的执行结果完全可以预测。要给用户猜测的数列，总是按照 `Fibonacci`、`Lucas`、`Pell`……的次序出现。我们希望让数列的出现顺序随机化（*randomize*）。这可以通过 C 语言标准库中的 `rand()` 和 `srand()` 两个函数达成：

```
#include <cstdlib>

srand( seq_cnt );
seq_index = rand() % seq_cnt;
current_vec = seq_addrs[ seq_index ];
```

`rand()` 和 `srand()` 都是标准库提供的所谓伪随机数（pseudo-random number）生成器。`srand()` 的参数是所谓随机数生成器种子（seed）。要知道，每次调用 `rand()`，都会返回一个介于 0 和“int 所能表示的最大整数”间的一个整数。现在我们必须将该值限制在 0~5，以便成为本例的一个有效索引。这两个函数的声明位于 `cstdlib` 头文件。

使用 `class object` 的指针，和使用内置类型的指针略有不同。这是因为 `class object` 关联了一组我们可以调用（*invoke*）的操作（*operation*）。举例来说，欲检查 `fibonacci vector` 的第二个元素是否为 1，我们可能会这么写：

```
if ( ! fibonacci.empty() &&
    ( fibonacci[1] == 1 ) )
```

如何才能间接通过 `pv` 达到同样的效果呢？上例中的 `fibonacci` 和 `empty()` 之间的句点，称为 *dot* 成员选择运算符（*member selection operator*），用来选择我们想要进行的操作。如果要通过指针来选择操作，必须改用 *arrow*（而非 *dot*）成员选择运算符：

```
! pv->empty()
```

由于指针可能并未指向任何对象，所以在调用 `empty()` 之前，应该先检验 `pv` 是否为非零值：

```
pv && ! pv->empty()
```

最后，如果要使用下标运算符（*subscript operator*），我们必须先提领 `pv`。由于下标运算符的优先级较高，因此 `pv` 提领操作的两旁必须加上小括号：

```
if ( pv && ! pv->empty() && ( (*pv)[1] == 1 ) )
```

我将在第 3 章深入讨论 *Standard Template Library*（*STL*），并在第 6 章设计和实现二叉树（*binary tree*）时，回头讨论指针相关议题。[LIPPMAN98]的 3.3 节对于指针有更深入的讨论。

1.7 文件的读写

Writing and Reading Files

用户可能会一再执行这个程序。我们应该让用户的分数可以在不同的“会话（*session*）”累计使用。为了达到这个目的，我们必须：(1) 每次执行结束，将用户的姓名及会话的某些数据写入文件，(2) 在程序开启另一个会话时，将数据从文件中读回。让我们看看这要怎么办。

要对文件进行读写操作，首先得包含 `fstream` 头文件：

```
#include <fstream>
```

为了打开一个可供输出的文件，我们定义一个 `ofstream`（供输出用的 *file stream*）对象，并将文件名传入：

```
// 以输出模式开启 seq_data.txt
ofstream outfile( "seq_data.txt" );
```

声明 `outfile` 的同时，会发生什么事？如果指定的文件并不存在，便会有一个文件被产生出来并打开供输出使用。如果指定的文件已经存在，这个文件会被打开用于输出，而文件中原有的数据会被丢弃。

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：85013855，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。